Efficient Out-of-Order Execution of Guarded ISAs

Nathanaël Prémillieu, André Seznec

Efficient Out-of-order Execution of Guarded ISAs

Nathanael Prémillieu, André Seznec

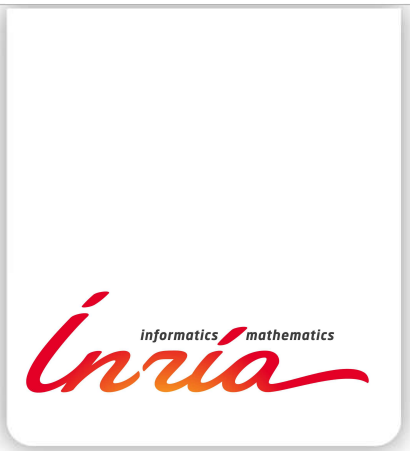
RESEARCH REPORT

N° 8406

28 novembre 2013

Project-Team ALF

ISSN 0249-6399 ISRN INRIA/RR--8406--FR+ENG



Efficient Out-of-order Execution of Guarded ISAs

Nathanael Prémillieu, André Seznec

Équipe-Projet ALF

Rapport de recherche n° 8406 — 28 novembre 2013 — 24 pages

Résumé : Les processeurs exécutant le jeu d'instructions ARM ne sont omniprésents. La demande de puissance de calcul est telle qu'aujourd'hui toutes les techniques jusqu'à prśent réservdes à la haute performance sont utilisées pour le design de ces processeurs.
Dans ce rapport, nous montrons que le jeu d'instruction prédiqué de ARM n'est pas un obstacle à la mise en oeuvre efficace de l'exécution dans le désordre.

Mots-clés : Architecture des processeurs

This work was partially supported by the European Research Council Advanced Grant DAL No 267175

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Efficient Out-of-order Execution of Guarded ISAs

Abstract: ARM ISA based processors are no longer low-cost low-power processors. Nowadays ARM ISA based processor manufacturers are struggling to implement medium-end to high-end processor cores which implies implementing a state-of-the-art out-of-order execution engine. Unfortunately providing efficient out-of-order execution on legacy ARM codes may be quite challenging due to guarded instructions.

Predicting the guarded instructions addresses the main serialization impact associated with guarded instructions execution and the multiple definition problem. Moreover guard prediction allows to use a global branch-and-guard history predictor to predict both branches and guards, often improving branch prediction accuracy. Unfortunately such a global branch-and-guard history predictor requires the systematic use of guard predictions. In that case, poor guard prediction accuracy would lead to poor overall performance on some applications.

Building on top of recent advances in branch prediction and confidence estimation, we propose a hybrid branch and guard predictor, combining a global branch history component and global branch-and-guard history component. The potential gain or loss due to the systematic use of guard prediction is dynamically evaluated at run-time. Two computing modes are enabled: systematic guard prediction use and high confidence only guard prediction use.

Our experiments show that on most applications, an overwhelming majority of guarded instructions are predicted. Therefore a simple but relatively inefficient hardware solution can be used to execute the few unpredicted guarded instructions. Significant performance benefits are observed on most applications while applications with poorly predictable guards do not suffer from performance loss.

Key-words: Processor, Architecture, Guard, Predicate, Prediction

Efficient Out-of-order Execution of Guarded ISAs

Nathanel Prémillieu and André Seznec

Abstract

ARM ISA based processors are no longer low-cost low-power processors. Nowadays ARM ISA based processor manufacturers are struggling to implement medium-end to high-end processor cores which implies implementing a state-of-the-art out-of-order execution engine. Unfortunately providing efficient out-of-order execution on legacy ARM codes may be quite challenging due to guarded instructions.

Predicting the guarded instructions addresses the main serialization impact associated with guarded instructions execution and the multiple definition problem. Moreover guard prediction allows to use a global branch-and-guard history predictor to predict both branches and guards, often improving branch prediction accuracy. Unfortunately such a global branch-and-guard history predictor requires the systematic use of guard predictions. In that case, poor guard prediction accuracy would lead to poor overall performance on some applications.

Building on top of recent advances in branch prediction and confidence estimation, we propose a hybrid branch and guard predictor, combining a global branch history component and global branch-and-guard history component. The potential gain or loss due to the systematic use of guard prediction is dynamically evaluated at run-time. Two computing modes are enabled : systematic guard prediction use and high confidence only guard prediction use.

Our experiments show that on most applications, an overwhelming majority of guarded instructions are predicted. Therefore a simple but relatively inefficient hardware solution can be used to execute the few unpredicted guarded instructions. Significant performance benefits are observed on most applications while applications with poorly predictable guards do not suffer from performance loss.

1 Introduction

Most instruction sets offer a limited form of guarded instructions, generally the conditional move, e.g. X86, Alpha, MIPS, SPARC V9, ... For these instruction sets, the compiler has limited option to generate if-converted branches [1], and in practice, the number of guarded instructions in the generated codes is quite limited. The impact of guarded instructions on the effective performance of the processor is also limited. On the other hand, other instruction sets such as ARM or IA64 have taken a much more radical approach : (nearly) all instructions can be guarded. Therefore the compiler has much more opportunity to generate guarded instructions.

The ARM-v7 ISA is now dominating the low power general-purpose processor segment. With the rise of mobile devices (smartphones, tablets), there is a constant demand for higher performance. Manufacturers are now adapting all the concepts that were used in high-end microprocessors to the ARM ISA, including out-of-order execution. However, providing efficient out-of-order execution

on a fully guarded ISA may be quite challenging [1].

The main difficulty for out-of-order execution of guarded instructions is the multiple definition problem [30]. This arises when the last instruction that may have written an architectural register R_i was a guarded instruction. In that case, when renaming the registers for a subsequent instruction I which uses R_i as an operand, one has to determine the effective physical register which will provide the value of R_i to instruction I, either the physical destination register of the guarded instruction or the old physical register associated with R_i. A working yet not efficient solution is to insert an extra non-architectural instruction after the guarded instruction [2]. This non-architectural instruction writes either the result of the operation of the guarded instruction or the old value depending on the dynamic guard (see Figure 4 in Section 2.3). However, this solution may hurt performance as it serializes the execution of possibly independent instructions e.g. when the same register is written on both paths of a branch that has been if-converted, thus reducing the available instruction level parallelism. More aggressive solutions [16, 6, 30] have been proposed to handle the multiple definition problem, but they induce a significant hardware overhead.

Predicting guarded instructions addresses the multiple definition problem [8]. However, systematic usage of guard prediction may sometimes lead to high guard misprediction rate and therefore to poor overall performance. Restricting the use of guard prediction to the high confidence predictions appears to limit performance degradation [17].

In this paper, we build on top of recent advances in branch prediction [23] and confidence estimation [21] for efficiently supporting out-of-order execution on a guarded ISA. We propose a hybrid branch and guard predictor, combining a global branch history predictor and a global branch-and-guard history predictor. This hybrid predictor will be referred to as the BO-BG predictor, for Branch Only history- Branch-and-Guard history. The BO-BG predictor is often more accurate on branch prediction than its branch-only history component. However, on some applications or application phases, the guard misprediction rate of BO-BG is quite high. In these cases, systematic use of guard predictions leads to lower performance.

Therefore we introduce a simple heuristic to dynamically estimate the performance benefit or loss of the systematic usage of guard prediction. Our BoL (for Benefit or Loss) heuristic determines whether to run in *systematic guard prediction use* mode or in *high confidence only guard prediction use* mode. When running in *high confidence only guard use* mode , the global branch-and-guard history can be corrupted with mispredicted guards, therefore, only the branch-only predictor component is used.

Our experiments using the BO-BG predictor and the BoL heuristic show that, on most applications, most guarded instructions are predicted. Therefore simple but relatively inefficient hardware solution can be used to execute the few unpredicted guarded instructions. Compared with out-of-order execution without guard prediction, significant performance benefits are encountered on most applications while applications with poorly predictable guards do not suffer any from performance loss. Moreover, our experiments also show that an

1. The handling of guarded instructions in currently implemented out-of-order execution processors is largely undocumented

aggressive implementation of guarded instruction execution is not worth the extra hardware complexity and power consumption.

The remainder of this paper is organized as follows. Section 2 provides background on the multiple definition problem in out-of-order execution processor using guarded ISAs. Section 3 presents related works on guarded instructions and guard prediction. Guards in the ARM-v7 ISA are described in Section 4. Section 5 details our BO-BG predictor proposal and the associated BoL heuristic. Section 6 presents our evaluation framework. Section 7 presents our experimental simulation results on ARM codes generated with a standard gcc compiler. Finally, Section 8 concludes this study.

Terminology

When referring to ISAs, guards and predicates are used as synonyms. To avoid repeating "predicate prediction" and "predicted predicate" in the paper, we will use the term guard apart in the expression "False Predicated Conditional Move" that was previously coined by Quiñones et al. [17].

2 Executing Guarded Instructions on an Out-of-order Engine

2.1 Register Renaming (no predication)

In an out-of-order execution engine, the mapping table is used to store the links between architectural registers and their associated physical registers value. This mapping table is used to avoid false dependencies between instructions that access the same architectural register. Hence, for each instruction, the rename stage assigns new physical registers to the architectural destination registers and the architectural source registers are renamed. A physical register P associated with architectural register R is considered as dead when the next write on R has been committed ; at this time it can be inserted in the free list and used again for renaming.

Figure 1 illustrates an example of the register renaming process. Instruction I reads from architectural registers $R1$ and $R2$, and writes into architectural register $R3$. To obtain the renamed form of instruction I, one has to read the mapping table. In this example, $R1$ is mapped to $P12$ and $R2$ to $P15$. The result register $R3$ is assigned to the first physical register available in the free list of physical register, $P22$ in this case. Then, the renamed form of I is $I : \ P22 \leftarrow P12, P15$.

All these steps are performed by the renaming stage before executing the instructions. Though renaming is applied to multiple instructions in parallel, the process preserves the in-order semantic of the program.

2.2 The Multiple Definition Problem on Out-of-execution Processors

When considering a guarded instruction, one cannot determine at the rename stage whether it will effectively write its architectural register target at write back or not because the guard value is often not known at this point.

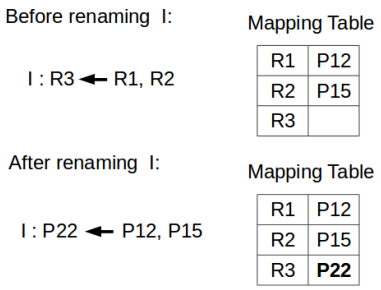


FIGURE 1 – Illustration of the register renaming process in an out-of-order processor.

Figure 2 illustrates this issue, known as the multiple definition problem [30]. I_1 conditionally writes to architectural register $R1$, I_1 being guarded with the guard p. After renaming, I_1 conditionally writes to $P1$. I_2 reads from $R1$, but it is not possible to know whether the correct physical register associated with $R1$ is $P1$ or $P11$ before the guard associated with I_1 is computed.

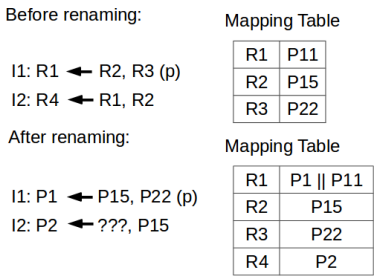


FIGURE 2 – The multiple definition problem on an out-of-order execution engine.

2.3 Dealing with the Multiple Definition Issue

2.3.1 False Predicated Conditional Moves

On an out-of-order execution processor, the execution of guarded instruction I writing the architectural register Res :

$(guard)\ ?\ Res\ \leftarrow Operation(Op1,\ Op2)$

should result at execution stage in :

$P_{after}\ =\ (guard)\ ?\ Operation(Op1,\ Op2) : P_{before}$

where P_{before} and P_{after} are respectively the physical registers assigned to architectural register Res before and after instruction I. That is if the guard is false, the instruction copies the value from the physical register previously allocated to Res to the newly allocated physical register.

Quiñones et al. [17] refer at this functionality as *False Predicated Conditional Move*, FPCM.

Direct implementations of FPCM are considered in the literature [16, 6] . Figure 3 illustrates the artificial dependency that is created by FPCM. Instruction I2 must be executed after instruction I1 and this, independently of the effective value of the guard.

The direct implementation of FPCM leads to very significant hardware complexity in the design. Every guarded instruction has an extra physical register operand. Therefore, extra complexity is added in most of the stages of the pipeline, particularly on the physical register file (extra read ports), on the bypass network and on operand tracking in the issue logic.

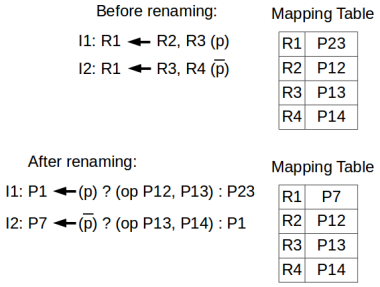


FIGURE 3 – False Predicated Conditional Move ; each guarded instruction is added an extra register operand : the old value of the target operand

2.3.2 Split FPCM

The above mentioned complexity can not be justified when the use of guarded instructions is quite infrequent, e.g. when the instruction set only allows conditional moves.

An alternative implementation consists in detecting the guarded instruction at decode time and splitting the instruction in two consecutive micro-operations : the first micro-operation executing the computation and the second micro-operation selecting between the previous target register value and the result of the first micro-operation. We will refer to this implementation as *split FPCM*[2]. The two micro-operations corresponding to the instruction I :

$(guard)\ ?\ Res\ \leftarrow Operation(Op1,\ Op2)$

are

$P_{new}\ =\ Operation(Op1,\ Op2)$

and

$P_{after}\ =\ (guard)\ ?\ P_{new} : P_{before}$

Figure 4 illustrates the serialization of the sequence of accesses on the registers, as well as the artificial creation of long dependency chains. Even if micro-operations $I1$ and $I2$ were executed on the same cycle T, the physical register P8 mapping architectural register R1 for subsequent instructions will not be valid before $I2'$ is executed (cycle $T+2$ at best). Such an implementation may impair performance when a significant amount of guarded instructions are executed.

2.3.3 Select-μ operation

The split FPCM mechanism inserts systematically an extra micro-operation for each guarded instruction. This results in a longer latency for guarded instruc-

2. Despites extensive search, we have not found any bibliographical reference on split FPCM. However this technique has been known in the industry since the mid 90's [2]

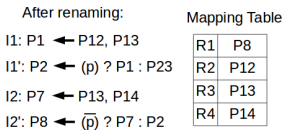


FIGURE 4 – Split FPCM : the first micro-operation unconditionally executes the operation, a second micro-operation selects between the new value or the old value.

tion as well as systematic serialization of potentially independent instructions.

Wang et al. [30] propose the select-μ op instruction, a solution to reduce this overhead and limit serialization to cases where it is mandatory. This instruction is conceptually similar to the ϕ-function used in the Single Static Assignment analysis [9]. The select-μ op is inserted just before a multiple definition must be resolved, e.g. when an non-guarded instruction reads a register that was last written by a guarded instruction.

Compared with split FPCM, select-μ op has several advantages. First, it postpones the insertion of the selection micro-operation till the effective use of the architectural register with a different guard. That is, in case of successive guarded writes on the same architectural register using the same guard, a single select-μ op selection instruction is inserted. Second, the (speculative) executions of two guarded instructions writing the same register, but guarded with opposite guards are not sequentialized and result in a single select-μ op insertion.

However, the select-μ op mechanism is rather complex to implement in hardware ; for instance each entry in the register mapping table must record two different physical register numbers and a physical guard register number. A single guarded instruction with two register operands and one guard can trigger up to three select-μ op insertions (one per register operand and one for the guard). Triggering these insertions requires to check the mapping table to look for the previous guarded definitions of the registers, adding extra complexity to the renaming stage. In comparison, the treatment of split FPCM can be implemented just at the exit of the decode stage.

Moreover, in their study, Wang et al. [30] fail to indicate how precise interruptions could be implemented : speculative registers allocated to an instruction may survive forever in the model proposed in [30] [3].

3 Related Works on Branch and Guard Predictions

Efficiently dealing with control instructions in a processor has always been a challenge. Two directions have been proposed, using prediction to know in advance the direction and the target of the branch [25], and using guarded instructions.

If-conversion was proposed by Allen et al. [1]. They define an algorithm to convert control dependencies into data dependencies by replacing branches and their dependent instructions by guarded instructions.The guarded instructions are only executed if their guard is evaluated to true. This conversion algorithm is

3. **Note for reviewers :** Solutions to solve this issue are out of the scope of our study

often called if-conversion. If-conversion allows to merge the taken and not-taken paths in the binary, it removes a branch, and allow to sequence both paths at the same time. However, it is not possible to if-convert all conditional branches. It is not always performance effective either, since both paths are fetched, increasing occupancy of the processor resources. Thus, one should only if-convert a subset of the convertible branches. Compilers often if-convert short branches only.

Combining branch prediction and guarded execution has been proposed in several studies [15, 14, 6]. The main idea is to have the compiler generating branch instructions and taken and not-taken paths for easy-to-predict control flow while hard-to-predict control flow is treated through if-conversion. This reduces the number of mispredicted branches at run-time and should increase performance. Chang et al. [6] use profiling to identify the hard-to-predict branches to convert. They show that profiling is efficient at identifying hard-to-predict branches. Kim et al. [15] further propose the wish branches. For each candidate to if-conversion, two versions of the code are generated, the guarded and the non-guarded code. Then, the executed version is chosen dynamically based on a confidence estimation.

Several studies [16, 29] point out that removing branches by if-conversion may impact the accuracy of branch prediction on the other branches. Simon et al. [24] observe that the outcomes of some branches can be directly related to the value of some guards, and therefore that applying if-conversion on selected branches often decreases branch prediction accuracy for other branches. They also propose to include guard information in the global branch history to try to capture the correlation lost by the if-conversion. However their approach is limited to include effective guard information when known at fetch time.

Guard prediction solves the multiple definition problem. Chuang and Calder [8] propose to use a guard predictor. The predictor is derived from a branch predictor. Contrary to branch prediction, on a guard misprediction, there is no need to squash the entire pipeline. Therefore the authors propose a selective replay mechanism, where only the instructions that depend on the mispredicted instruction are re-executed. Quiñones et al. [17] propose to selectively use the guard prediction. All guards are predicted, but the effective use of prediction is triggered on a per-guard basis using a confidence estimator. When the confidence is high enough, the prediction is used. If not, the guarded instruction is handled through False Predicate Conditional Moves. In a later work, Quiñones et al. [18] identify that branch outcomes as well as guard values are often correlated with former guard values and propose a combined branch and guard predictor. The predictor used in [18] is an alloyed local history/global branch-and-guard history perceptron predictor.

Our study is directly related to the work by Quiñones et al.. However, we identify that the association of the use of global branch-and-guard history with per-guard selective use of guard prediction can lead to the use of corrupted global branch-and-guard history. This further leads to significant performance loss when using state-of-the-art predictors such as TAGE [23], GEHL [19] or hashed perceptron [28]. This phenomenon is much less marked on a perceptron predictor as used in [18]. However the perceptron predictor is relatively inefficient compared with state-of-the-art, thus overall performance is disappointing (see Section 7).

4 Predicting Guards on the ARM ISA

Most of the previously published studies on out-of-order execution of guarded instruction ISA target the Intel Itanium ISA [12]. For this ISA, the guard of a guarded instruction is the boolean value contained in a guard register. In this study we use the ARM-v7 ISA that features some different specificities that we present below.

4.1 Guards on the ARM ISA

Instructions are guarded through a boolean value computed from the value of four flags : the Negative flag (N), the Zero flag (Z), the Carry flag (C) and the Overflow flag (V). These flags are written by specific instructions, like compare instructions and specific arithmetic instructions [3]. The values of the guards are not directly known through reading a register, but requires to evaluate a logical formula on the flags. Guards are paired with a guard and its opposite. Table 1 illustrates the set of possible guards. Pairs of opposite guards are grouped in the same entry, the logical formula corresponding to the first guard.

On ARM v7 ISA, conditional branches are guarded instructions.

guard Mnemonics	**Logical Formula**
(EQ,NE)	Z == 1
(CS,CC)	C == 1
(MI,PL)	N == 1
(VS,VC)	V == 1
(HI,LS)	(C == 1) && (Z == 0)
(GE,LT)	N == V
(GT,LE)	(Z == 0) && (N == V)
AL	(always true)

TABLE 1 – Possible guards for guarded instructions on ARM v7

4.2 Predicting Guarded Instructions

In many cases, several guarded instructions are using the same guard or the opposite guard — the original taken and not-taken paths. This leads to the concept of guarded group of instructions, i.e., the group of guarded instructions that use the same guard value or its opposite. A guarded group is associated with the use of the same occurrence of a specific guard. These instructions are not necessarily contiguous in the code since if-conversion leads to encapsulate and schedule instructions from the taken path, instructions from the not-taken path and instructions common to both paths in the same basic block. Figure 5 illustrates an example of two guarded groups. In our study, a guarded group starts at the first use of a guard and ends when a flag-defining instruction is encountered.

When guard prediction is used, one has only to predict the guard for the first instruction of the guarded group. In the remainder of the paper, when referring to the global branch-and-guard history vector, we assume that the guard is

appended only once to the history on its first encountering in the instruction flow, even when the same guard is used multiple times.

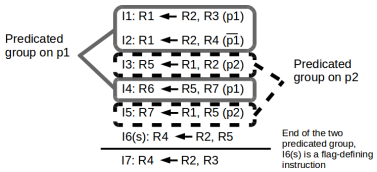


FIGURE 5 – A guarded group is a group of instructions that use the same occurrence of a guard or its opposite. It ends when a flag-defining instruction is encountered.

5 Branch and Guard Prediction

5.1 Branch History vs Branch-and-Guard History

The accuracy of a branch or a guard predictor depends on the prediction scheme, the predictor size and also on the quality of the information that the predictor is exploiting. Global branch or path history is generally considered as the highest quality information usable to predict branches. All recent branch predictor proposals such as TAGE [23], GEHL [19], hashed perceptron [28] or SNAP [27] used global branch or path history as their main input vector. Adjunct predictors using other information inputs, such as the loop predictor [10, 20] or a local history predictor component [13, 21] may bring some extra but relatively marginal accuracy benefit.

Several studies [29, 18] have pointed out that the global branch-and-guard history is often a better information vector than the global branch history, since branches are often correlated to some guards. Therefore, one would like to use the global branch-and-guard history predictor to predict both the branches and the guards for guarded instructions.

Branch and guard predictors are accessed at prediction time with a speculative history and updated at commit time with a non-speculative history. The speculative global branch history used to read a prediction matches exactly the commit time global branch history on the right path. If all guards are predicted and the pipeline is flushed on every guard misprediction then the same applies for global branch-and-guard history.

However, systematically using guard prediction can lead to a performance loss compared to the use of split FPCM (see Section 7). Therefore, selective use of guard prediction as proposed in [17] is appealing. For instance, one may only use guard prediction when the confidence of the prediction is high [17]. In that case, a low confidence guard misprediction does not result in a pipeline flush, but it results in a corrupted speculative global branch-and-guard history. The branches and the guards predicted after the mispredicted guard are predicted using a wrong global branch-and-guard history.

On most predictors, a corrupted global branch-and-guard history induces reading wrong entries on the predictor, as illustrated in Figure 6. E.g. on TAGE or GEHL, for the predictions just following the mispredicted guard, all the tables are read with a wrong entry number. The perceptron predictor is much less

sensitive to this corruption, as illustrated in Figure 7. Since all predictor counters are accessed using only the program counter, only the counter associated with the mispredicted guard corrupts the prediction : If the predicted branch or guard is not strongly correlated with the mispredicted guard then the absolute value of the counter will be small and the prediction result is likely to be unaffected.

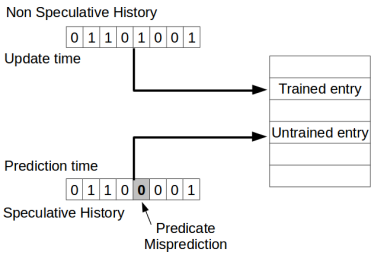


FIGURE 6 – Corrupted branch-and-guard history leads to read a wrong predictor entry

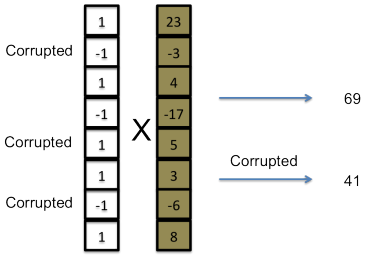


FIGURE 7 – Corrupted branch-and-guard history on a perceptron predictor does not often mean incorrect prediction

5.2 The Branch-Only-Branch-and-Guard Predictor

To address the issue of corrupted branch-and-guard history mentioned above, we propose BO-BG (Figure 8) a hybrid predictor consisting in a global branch history component, BO, and a global branch-and-guard history component, BG. The predictor is used at fetch time to predict the branches and guards in the fetch group. A meta predictor, META, **and** a Benefit-or-Loss heuristic hardware mechanism, BoL, are used to choose among the predictions flowing out from the two components.

The benefit-or-loss heuristic hardware mechanism, BoL, determines the execution mode : Either *systematic guard prediction use* mode (SY-mode), or *high confidence only guard use* mode (HCO-mode). When running in SY-mode, all the guards are predicted and the predictions are systematically used ; mispredictions are resolved in the execution stage and fetch is resumed at the first instruction of the guarded instruction group. That is, when executing on the correct path, the speculative branch-and-guard history used at prediction time is the correct branch-and-guard history. Therefore one can use the predictions (for branches and guards) that flow from both the BO and BG components.

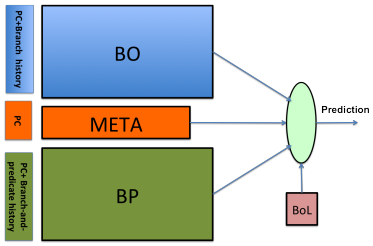


FIGURE 8 – The BO-BG branch and guard predictor

On the other hand, when running in HCO-mode, some guards are not predicted and the speculative branch-and-guard history is potentially corrupted. Therefore one should use only the predictions flowing out from the BO component.

BoL is in charge of determining whether to run in SY-mode or HCO-mode. For this, we use a simple yet efficient heuristic. In both modes, the BO, BG and META components are systematically updated at commit time as if the processor was running in SY-mode. BoL uses a simple signed 11-bit saturated counters which is updated according to Algorithm 1.

Algorithm 1 The Benefit-or-Loss heuristic

```
if Branch then
  if Pred(BO-BG) ≠ Pred(BO) then
    BoL += (Pred(BO-BG) correct) ? Penalty : -Penalty
  end if
end if

if Non-branch guard then
  if Pred(BO) not high confidence then
    BoL += (guarded group size)
    if Pred(BO-BG) incorrect then
      BoL -= Penalty
    end if
  else
    if Pred(BO-BG) ≠ Pred(BO) then
      BoL += (Pred(BO-BG) correct) ? Penalty : -Penalty
    end if
  end if
end if
```

The intuition behind the BoL heuristic is that 1) the performance benefit from a correct prediction of a guard is approximately proportional to the size of the guarded group 2) the performance loss (resp. benefit) from an extra misprediction (resp. extra correct prediction) can be modeled by an average penalty.

Switching from HCO-mode to SY-mode implies restoring a correct speculative branch-and-guard history, i.e., draining the complete pipeline. To avoid ping-ponging back and forth between HCO-mode and SY-mode, the HCO-mode (resp. SY-mode) is triggered only when the BoL counter becomes lower than -512 (resp. higher than 512).

OoO@1GHz	4-way	8-way
Memory	100 cycles 12.8GBps, Across a 128B bus	
Caches	L1D 4-way 64KB, 64B, 1 cycle L1I 4-way 64KB, 64B, 1 cycle L2 8-way 4MB, 64B, 8 cycles Stride Prefetcher for the L2	
TLBs	Perfect, 4K pages	
ROB	128 entries	256 entries
IQ	64 entries	128 entries
LSQ	128 entries (64L/64S)	256 entries (196L/64S)
Width (F/D/R/I/E/W/C)	4	8
Pipeline	12 stages	
FU(latency) IntAlu(1) IntMultDiv(3/12*) FpAlu(5) FpMultDiv(4/9*) Ld(2) Str(1)	 3 2 2 2 2 2	 6 2 4 4 2 2
Instruction fetch	BTB 4-way, 1K entries RAS, 16 entries, WP corruption detection 2 x TAGE 1+12 components, 15 Kentries META, 1 Kentries, 5-bit Perceptron, 40-bit history, 1 Kentries	
Misprediction penalty	15 cycles	15 cycles

TABLE 2 – Simulator configuration overview. *not pipelined.

In the remainder of the paper, *Penalty* is an empirically determined constant. However, the best value for *Penalty* would depend on the precise core micro-architecture (issue width, pipeline depth, ..). It can also dynamically depend on the application and on the application phase. Adaptive *Penalty* is left for future exploration.

In the remainder of the paper, the BG and BO components of the BO-BG predictor will be TAGE predictors enhanced with a storage-free confidence mechanism close to the one described in [22]. For non-branch guards and on a correct prediction provided by a counter which value is 1,2, -2 or -3, the prediction counter is incremented with probability $\frac{1}{32}$. This small modification allows to avoid many high confidence mispredictions without significantly modifying the global misprediction rate. 256 Kbits storage budgets are considered for each of the TAGE components, and a PC indexed 1024 5-bit entries META predictor is modeled.

Other global history predictors can also be considered e.g., GEHL [19], Hashed perception [28] or SNAP [27]. Global history perceptron predictors will also be considered in Section 7.1.2, since they present the particularity of being quite resilient to branch-and-guard history corruption. We do not consider any local branch (or guard history) component as their extra accuracy contribution extra accuracy is marginal while their hardware implementation is quite tricky ; in particular predicting several branches and several guards per cycle with a local predictor, maintaining speculative local histories involve complex hardware logic.

Benchmarks	inputs	**IPC (BASE)**		**% guarded**	
		4-way	**8-way**	**with branches**	**without branches**
400.perlbench	checkspam diffmail	1.47	1.73	16.59	4.63
401.bzip2	chicken combined liberty program source text	2.21	2.7	16.46	4.68
403.gcc	c-typeck 166 cp-decl expr scilab 200	1.67	2.15	36.11	24.47
416.gamess	cytosine h2ocu2+	3.17	4.84	6.7	2.89
429.mcf	ref	0.71	0.81	24.5	6.1
435.gromacs	ref	3.01	4.8	4.62	1.02
436.cactusADM	ref	3.53	4.3	0.09	0.02
444.namd	ref	2.62	3.83	8.54	5.18
445.gobmk	13x13 nngs trevorc trevord	1.8	2.29	18.05	6.86
453.povray	ref	1.81	2.35	8.09	1.9
456.hmmer	nph3 retro	3.25	4.87	17.41	14.67
458.sjeng	ref	1.82	2.3	18.76	7.05
459.GemsFDTD	ref	2.53	3.89	1.06	0.001
462.libquantum	ref	1.86	2.11	23.56	13.42
464.h264ref	sss baseline main	2.57	3.21	6.4	2.41
470.lbm	ref	2	2.34	0.6	0.02
471.omnetpp	ref	0.81	0.91	17.1	4.33
473.astar	BigLakes rivers	1.4	1.76	15.62	3.59
483.xalancbmk	ref	1.83	2.45	21.09	3.4

TABLE 3 – Benchmarks, their inputs, their IPC for the BASE 4-way and 8-way configurations and the ratio of guarded instructions over the total number of instructions.

6 Experimental Framework

The experimental study for validating our propositions was built upon the Gem5 simulator [5].

6.1 Simulator Parameters

The simulator models an aggressive 4-way superscalar processor. Split FPCM is modeled for guarded instructions when guard prediction is not used. The processor also features a state-of-the-art conditional branch predictor, the TAGE predictor described in [23]. The store sets predictor [7] is used to predict memory dependencies.

The other characteristics are summarized in Table 2.

The BASE configuration is the configuration **without** guard prediction and featuring only the BO branch predictor component.

We report simulation results (speed-ups over the BASE configuration) assuming a 4-way superscalar processor, except in Section 7.6 which shows that trends are amplified for a 8-way superscalar processor.

6.2 Benchmarks

The simulated benchmarks constitute a subset of the Spec 2006 benchmarks set [26] listed in Table 3. To reduce the amount of simulation time, we use the Simpoint methodology [11] to summarize each benchmark in a set of 100 millions instructions slices. Each slice is representative of a part of the benchmark execution and is affected a weight representing the portion that it represents in the execution. For each benchmark, the illustrated results are the weighted mean of simulations on the set of slices [11]. Table 3 displays the weighted mean

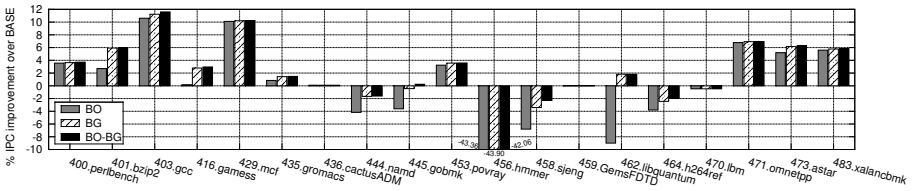


FIGURE 9 – Systematic guard prediction use : TAGE-based predictor

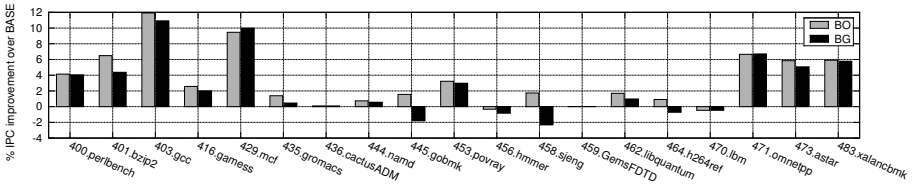


FIGURE 10 – High confidence only guard prediction use : TAGE-based predictor

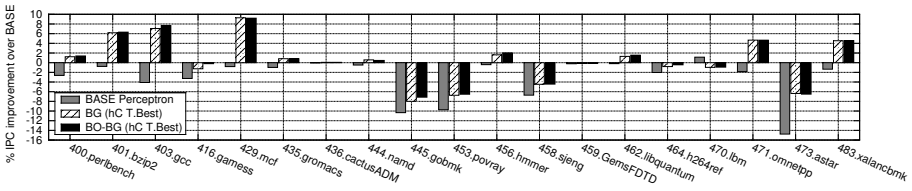


FIGURE 11 – High confidence only guard prediction use : perceptron-based predictor

of the Instruction Per Cycle (IPC) count for each benchmark, for the 4-way and 8-way BASE configurations.

As we target the ARM instruction set, some of the benchmarks or some of their input sets are missing. There are three reasons why some benchmarks are missing : 1) the binary produced by our cross-compiler is not executable on a native ARM architecture, 2) the binary is not executable on *qemu-arm* [4] which was used to compute the basic block vector (BBV) needed to compute the simpoints 3) the Gem5 ARM-v7 simulator is not able to run them. In the end, we were able to run 12 integer benchmarks (the complete set of integer benchmarks) and 7 floating point benchmarks. Some benchmarks are used with several inputs (all the inputs that are working are used). In total, we were able to simulate 38 different workloads. For each benchmark, the results showed are the average results of its different inputs.

The binaries were generated with the gcc compiler using the O3 optimization level. Gcc decision to if-convert a branch mainly depends on the number of instructions that are controlled by the branch. By default, for the ARM target, this number is set to 4.

6.3 Ratio of Guarded Instructions

Table 3 also lists the ratio of guarded instructions per benchmark. The first column presents the total percentage of guarded instructions. This includes the conditional branches. The second column excludes conditional branches.

For all benchmarks, conditional branch instructions represent a large part of the guarded instructions. However, some benchmarks like *401.bzip2*, *403.gcc*, *445.gobmk* and *456.hmmer* contain a significant portion of effective guarded instructions. Some other benchmarks, like *436.cactusADM*, *459.GemsFDTD* and *470.lbm* feature nearly no effective guarded instructions.

A simple optimization to save energy would be to monitor at run-time the ratio of effective guarded instructions and to turn-off the guard prediction when this ratio is under a predefined threshold.

7 Experimental results

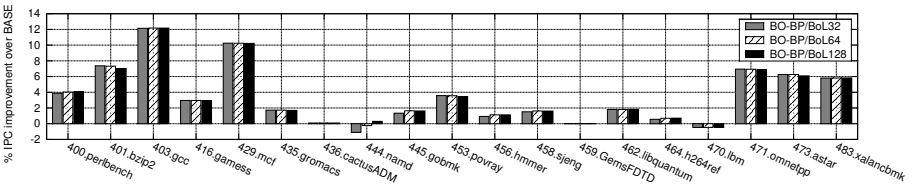


FIGURE 12 – Impact of the *Penalty* constant in BoL heuristic

7.1 Branch History versus Branch-and-Guard History

Unless explicitly mentioned, simulation results are reported in relative speed-ups over the BASE 4-way configuration.

7.1.1 Systematic Guard Prediction Use

Figure 9 reports simulations results assuming that guard predictions are systematically used for 3 predictors : a BO-history TAGE, a BG-history TAGE and the BO-BG predictor without using the BoL heuristic.

As expected, systematic guard prediction is effective at enabling performance gain on most applications. But on some applications e.g. *444.namd*, *456.hmmer*, *458.sjeng*, *464.h264ref*, ... performance losses are encountered. The performance loss is even dramatic on *456.hmmer*. Therefore systematic guard prediction use should not be considered for implementation in real hardware. One can also remark that using branch-and-guard history is often beneficial, e.g. on *401.bzip2*, *416.games* or *462.libquantum*, but not systematically, e.g. on *429.mcf* or *471.omnetpp*. As expected the BO-BG predictor slightly outperforms its two components.

7.1.2 High Confidence Only Guard Prediction Use

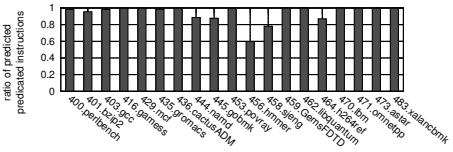


FIGURE 13 – Use of predicted guarded instructions

Figure 10 illustrates simulations results assuming that guard predictions are used only when high confidence. As anticipated, the BG-history TAGE results most often in lower performance than the BO-history TAGE, because the speculative branch-and-guard history is corrupted by incorrect guard predictions. Restricting the prediction usage to high confidence branches appears as an effective filter to eliminate performance loss due to guard mispredictions for the BO predictor. BO-history TAGE should even be considered as a valid design point for effective designs, since it outperforms the BASE design at the exception of a marginal loss on *470.lbm*.

The BO-BG predictor (not illustrated) is not worth, as a design point since its BG component has poor behavior.

We run similar simulations assuming perceptron predictors as base components instead of TAGE. We assume 40 bits history length and 1 Kentries predictors, i.e. a 41 Kbytes perceptron predictor. As mentioned in Section 5.2, the perceptron predictor should be much more resilient to branch-and-guard history corruption than TAGE. Figure 11 reports results for this experiment. Prediction confidence is estimated as follows. An extra counter is added to each perceptron entry to monitor the correctness of the predictions. This confidence counter is incremented on a correct prediction, and reset on a misprediction. High confidence is considered on saturated counters only. As the perceptron predictor is not our main target we run multiple simulations varying the counter width from 0 to 7 bits, and we only illustrate the best configuration for each benchmark. The reported results should therefore be considered as an upper limit.

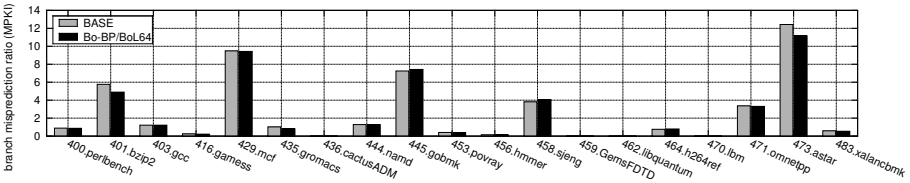


FIGURE 14 – Branch misprediction rates

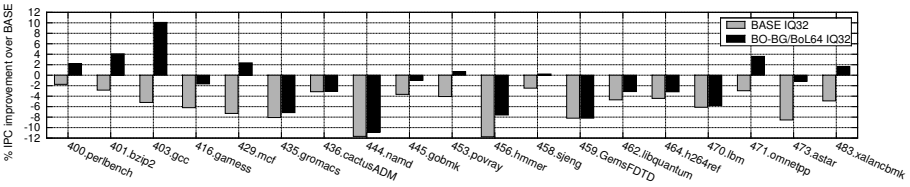


FIGURE 15 – Instruction Queue size impact

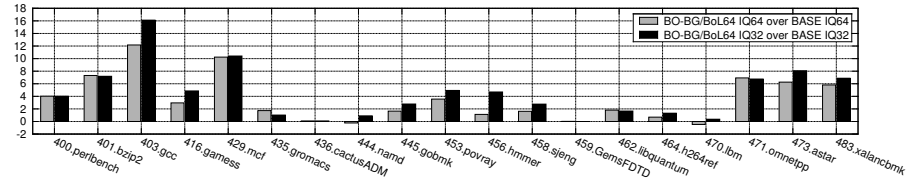


FIGURE 16 – Relative guard prediction benefit for different Instruction Queue sizes

First the perceptron predictor alone without guard prediction is quasi-systematically outperformed by the TAGE predictor, and often by a quite significant margin. As reported by Quiñones et al. [18], branch-and-guard history associated with high confidence only guard use allows to systematically outperform the perceptron predictor without suffering major performance loss on any benchmark. However, the benefit is limited and the performance is lower than BO-history TAGE + high confidence only guard prediction use. The performance often does not reach the level of our BASE using a TAGE predictor without any guard prediction use.

7.2 BO-BG Predictor with BoL Heuristic

Figure 12 reports simulation results assuming the BO-BG TAGE predictor assuming respectively 32, 64 and 128 as the *Penalty* constant in the BoL heuristic. As expected, BO-BG/BoL allows to reach performance higher than just running in SY-mode or just running in HCO-mode. It also slightly outperforms the best of the two modes for all benchmarks.

However the performance impact of the *Penalty* constant is relatively low. This tends to indicate that in most code sections, one of the two modes SY-mode or HCO-mode has a clear performance benefit over the other mode.

In the remainder of the paper, we will assume that *Penalty*=64.

7.3 Performance Analysis

The performance benefit allowed by our proposal comes from three different factors. First, guard prediction eliminates the execution of the extra instruction in split FPCM and eliminates the execution of the predicted guarded instructions which guard is predicted false. Second it greatly simplifies the dependency chain, eliminating the artificial data dependency created by by guarded execution and for predicted true guarded instructions breaking the dependency chain with the guard writer.

Figure 13 illustrates the ratio of predicted guarded instructions used in the different benchmarks. On many applications, the most significant part of the application runs in SY-mode. Even on applications running essentially in HCO-mode, a very significant part of the guards are predicted with high confidence with a minimum of 60 % on *456.hmmer*.

Second, the overall conditional branch misprediction rate is sometimes significantly reduced through using a hybrid predictor using both branch history and branch-and-guard history as illustrated in Figure 14, e.g. on *401.bzip2* or *473.astar*.

7.4 Reducing the Instruction Queue Pressure

By using guard prediction, the number of instructions that enter the instruction queue is substantially reduced. First for predicted false guard instructions, only the first instruction in the guarded group enters the instruction queue. Second the predicted guarded instructions are not split.

Figure 15 illustrates the performance assuming a 32-entry instruction queue instead of a 64-entry instruction queue. Without guard prediction, the impact of instruction queue size reduction is important on a few benchmarks, e.g *444namd,*

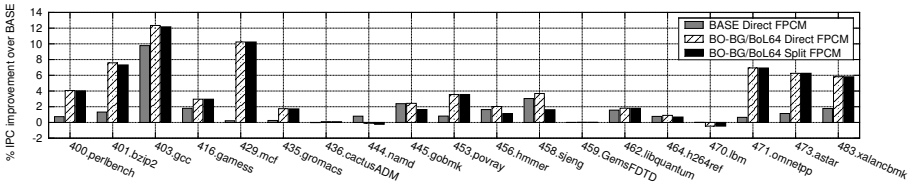


FIGURE 17 – Split FPCM versus direct FPCM

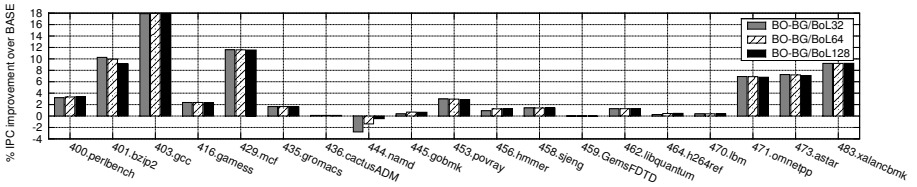


FIGURE 18 – Guard prediction on a 8-way issue superscalar processor. BASE is 8-way

456.hmmer and *473.astar*. Figure 16 illustrates that the relative benefit of using guard prediction is generally higher when the instruction queue size is smaller, e.g. *403.gcc* and *456.hmmer*.

7.5 Split FPCM vs FPCM

Up to now we have assumed that split FPCM execution is used for guarded instructions. Direct implementation of FPCM would save an operation per guarded instruction (if guard prediction is not used), but would require an extra register operand per guarded instruction (see Section 2.3). Since in most cases, guarded instructions are predicted, the potential benefit of a direct implementation of FPCM can be much lower when guard prediction is used.

Figure 17 illustrates the performance benefit of using FPCM instead of split FPCM. There is a small performance benefit when guard prediction is not used, less than 3 % except on *403.gcc* up to 10 %. This benefit vanishes when guard prediction is used through our BO-BG/BoL proposal. This is expected since for most benchmarks, most of the guarded instructions are predicted.

Therefore, the extra hardware complexity associated with the direct implementation of FPCM instead of split FPCM is not worth paying when guard prediction is used through our BO-BG/BoL predictor.

7.6 Wide Issue Superscalar Processor

The benefit of our BO-BG/BoL proposal is growing when one considers a more aggressive implementation featuring a wide issue out-of-order engine. Figure 18 illustrates this on a 8-way superscalar processor. 64 appears as a correct tradeoff for the BoL *Penalty* constant. The speedup over a 8-way base superscalar grows to up to 18% on *403.gcc* and the relative speedup is most

often higher for 8-way issue than for 4-way issue.

8 Conclusion

ARM based processors are becoming ubiquitous in many modern appliances including smartphones and tablets. The demand for high performance pushes manufacturers of ARM processors to use the same techniques that have been used for the two last decades on PCs and server processors including wide-issue superscalar processors. The 32-bit ARM v7 ISA features guarded instructions. Providing an efficient solution to efficiently execute guarded instructions out-of-order is challenging due to the multiple definition problem.

Fortunately, predicting the guarded instructions addresses the multiple definition problem. In this paper, we have shown that state-of-the-art global branch history predictor can be adapted to predict both branches and guards. We have proposed BO-BG, a hybrid predictor combining a branch-only history component and a branch-and-guard component. Unfortunately, systematic use of guard prediction and sometimes poor overall guard prediction accuracy leads to poor overall performance, sometimes significantly worse that the performance without guard prediction.

As such, we have proposed BoL, a simple heuristic that evaluates dynamically the potential gain associated with systematic use of guard prediction. BoL is used to control two execution modes, systematic guard prediction use or SY-mode and high confidence only guard prediction use or HCO-mode.

Our experiments show that the association of BO-BG with BoL allows to achieve high out-of-order execution performance on a guarded instruction set. The processor runs in SY-mode on code sections where the guards are highly predictable and/or branch-and-guard history allows to reduce the conditional misprediction rate. The processor runs in HCO-mode on regions where the guard misprediction rate is high, but still a significant number of guards are predicted since their prediction is high confidence. Therefore, the simple but relatively inefficient hardware associated with split false predicated conditional move is sufficient for executing the few non predicted guarded instructions. Significant performance benefits are encountered on most applications while applications with poorly predictable guards do not suffer significant performance loss. This benefit will be even higher on future very wide-issue superscalar processors.

Therefore guarded ISA is not an obstacle anymore for the implementation of efficient out-of-order execution. BO-BG/BoL could be considered as an opportunity to allow more aggressive use of guarded instructions by the compiler. For instance, on applications featuring a few suspected poorly predictable branch statements inside loops, e.g *456.hmmer*, the branch control-flow instructions could be handled through if-conversion. At runtime, depending on the global guard predictability, the hardware will decide whether to execute them either in HCO-mode or in SY-mode.

Acknowledgement

This work was partially supported by the European Research Council Advanced Grant DAL No 267175

Références

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1983, pp. 177–189.

[2] "Alpha 21264 microprocessor hardware reference manual," Compaq Computer Corporation, 1999.

[3] ARM, "Arm architecture reference manual. arm v7-a and arm v7-r edition."

[4] F. Bellard, "QEMU," *http://wiki.qemu.org/Main_Page*.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[6] P.-Y. Chang, E. Hao, Y. N. Patt, and P. P. Chang, "Using predicated execution to improve the performance of a dynamically scheduled machine with speculative execution," *International Journal of Parallel Programming*, vol. 24, no. 3, pp. 209–234, 1996.

[7] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *ISCA '98 : Proceedings of the 25th annual international symposium on Computer architecture*, 1998, pp. 142–153.

[8] W. Chuang and B. Calder, "Predicate prediction for efficient out-of-order execution," in *Proceedings of the 17th annual international conference on Supercomputing*, 2003, pp. 183–192.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, 1991.

[10] H. Gao and H. Zhou, "Adaptive information processing : An effective way to improve perceptron predictors," *Journal of Instruction-Level Parallelism*, vol. 7, 2005.

[11] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0 : Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. vol. 7, September 2005.

[12] Intel Corp, "Intel itanium architecture software developerâĂŹs manual. volume 3 : Instruction set reference," 2002.

[13] D. A. Jiménez and C. Lin, "Neural methods for dynamic branch prediction," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 369–397, 2002.

[14] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt, "Diverge-merge processor (dmp) : Dynamic predicated execution of complex control-flow graphs based on frequently executed paths," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006, pp. 53–64.

[15] H. Kim, O. Mutlu, J. Stark, and Y. N. Patt, "Wish branches : Combining conditional branching and predication for adaptive predicated execution," in *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, 2005, pp. 43–54.

[16] D. N. Pnevmatikatos and G. S. Sohi, "Guarded execution and branch prediction in dynamic ilp processors," in *Proceedings of the 21st annual international symposium on Computer architecture*, ser. ISCA '94, 1994, pp. 120–129.

[17] E. Quiñones, J.-M. Parcerisa, and A. Gonzalez, "Selective predicate prediction for out-of-order processors," in *Proceedings of the 20th annual international conference on Supercomputing*, 2006, pp. 46–54.

[18] E. Quinones, J.-M. Parcerisa, and A. Gonzailez, "Improving branch prediction and predicated execution in out-of-order processors," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, ser. HPCA '07. Washington, DC, USA : IEEE Computer Society, 2007, pp. 75–84. [Online]. Available : http://dx.doi.org/10.1109/HPCA.2007.346186

[19] A. Seznec, "Analysis of the O-GEometric History Length branch predictor," in *ISCA*, 2005, pp. 394–405.

[20] ——, "The L-TAGE branch predictor," in *Journal of Instruction Level Parallelism*, May 2007.

[21] ——, "A new case for the tage branch predictor," in *MICRO*, 2011, pp. 117–127.

[22] ——, "Storage free confidence estimation for the tage branch predictor," in *HPCA*, 2011, pp. 443–454.

[23] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *Journal of Instruction Level Parallelism*, February 2006.

[24] B. Simon, B. Calder, and J. Ferrante, "Incorporating predicate information into branch predictors," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003, pp. 53–64.

[25] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th annual symposium on Computer Architecture*, 1981, pp. 135–148.

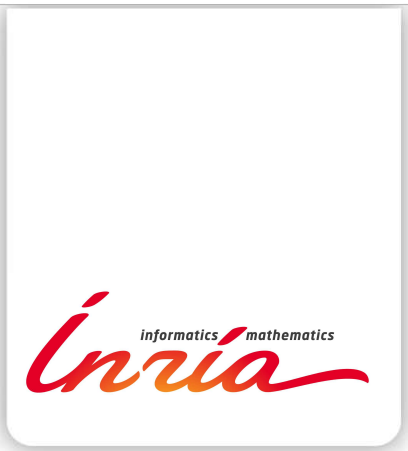
[26] SPEC, "SPEC CPU2006," *http://www.spec.org/cpu2006/*, 2006.

[27] R. St. Amant, D. A. Jimenez, and D. Burger, "Low-power, high-performance analog neural branch prediction," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41. Washington, DC, USA : IEEE Computer Society, 2008, pp. 447–458. [Online]. Available : http://dx.doi.org/10.1109/MICRO.2008.4771812

[28] D. Tarjan and K. Skadron, "Merging path and gshare indexing in perceptron branch prediction," *TACO*, vol. 2, no. 3, pp. 280–300, 2005.

[29] G. S. Tyson, "The effects of predicated execution on branch prediction," in *Proceedings of the 27th annual international symposium on Microarchitecture*, 1994, pp. 196–206.

[30] P. H. Wang, H. Wang, R.-M. Kling, K. Ramakrishnan, and J. P. Shen, "Register renaming and scheduling for dynamic execution of predicated code," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001, pp. 15–25.



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399